SYSTÈME D'EXPLOITATION

Gestion de la mémoire

CHAPITRE 2
GESTION DE LA MÉMOIRE

INTRODUCTION :

Loi de Moore: *les capacités de stockage doublent en taille tous les 18 mois.*

Loi de Parkinson: *les programmes grossissent en taille aussi vite que la mémoire.*

→ On aura toujours besoin de gestionnaires de mémoires performants.

Le Gestionnaire de la Mémoire gère l'hiérarchie de mémoire (allouer, libérer, transfert …)

- Mémoire du cache: volatile, rapide, chère
- Mémoire centrale: volatile, moins rapide, moins chère
- Mémoire de masse –disque: non volatile, lente, pas chère

GESTION ÉLÉMENTAIRE

Monoprogrammation:

- Mémoire réservée au SE
- Mémoire réservée au seul processus en exécution.

Multiprogrammation:

- La Multiprogrammation améliore le taux d'utilisation du CPU mais requiert un bon partitionnement de la mémoire.
- A l'initialisation du système, la mémoire est divisée en *n* partitions de taille fixe.
- De préférence, des partitions inégales en taille.

CONCEPTS LIÉS À LA GESTION DE LA MÉMOIRE

- Le gestionnaire de mémoire:

Le gestionnaire de mémoire a pour rôle de :

- Allouer de la mémoire aux processus ;
- Connaître les zones mémoire libres ou occupées ;
- Récupérer de la mémoire en fin d'exécution ;
- Traiter le va-et-vient entre le disque et la mémoire centrale (swap).

CONCEPTS LIÉS À LA GESTION DE LA MÉMOIRE

Chargement de processus en mémoire

- Un programme doit être placé en mémoire centrale (RAM) afin d'être exécuté (sous forme de processus) ;
- Ce chargement en mémoire prend du temps ;
- Un processus se trouve souvent interrompu au cours de son exécution, par exemple en attente d'une Entrée/Sortie ;
- Le système a l'objectif de conserver le plus grand nombre de processus actifs en mémoire.

CONCEPTS LIÉS À LA GESTION DE LA MÉMOIRE

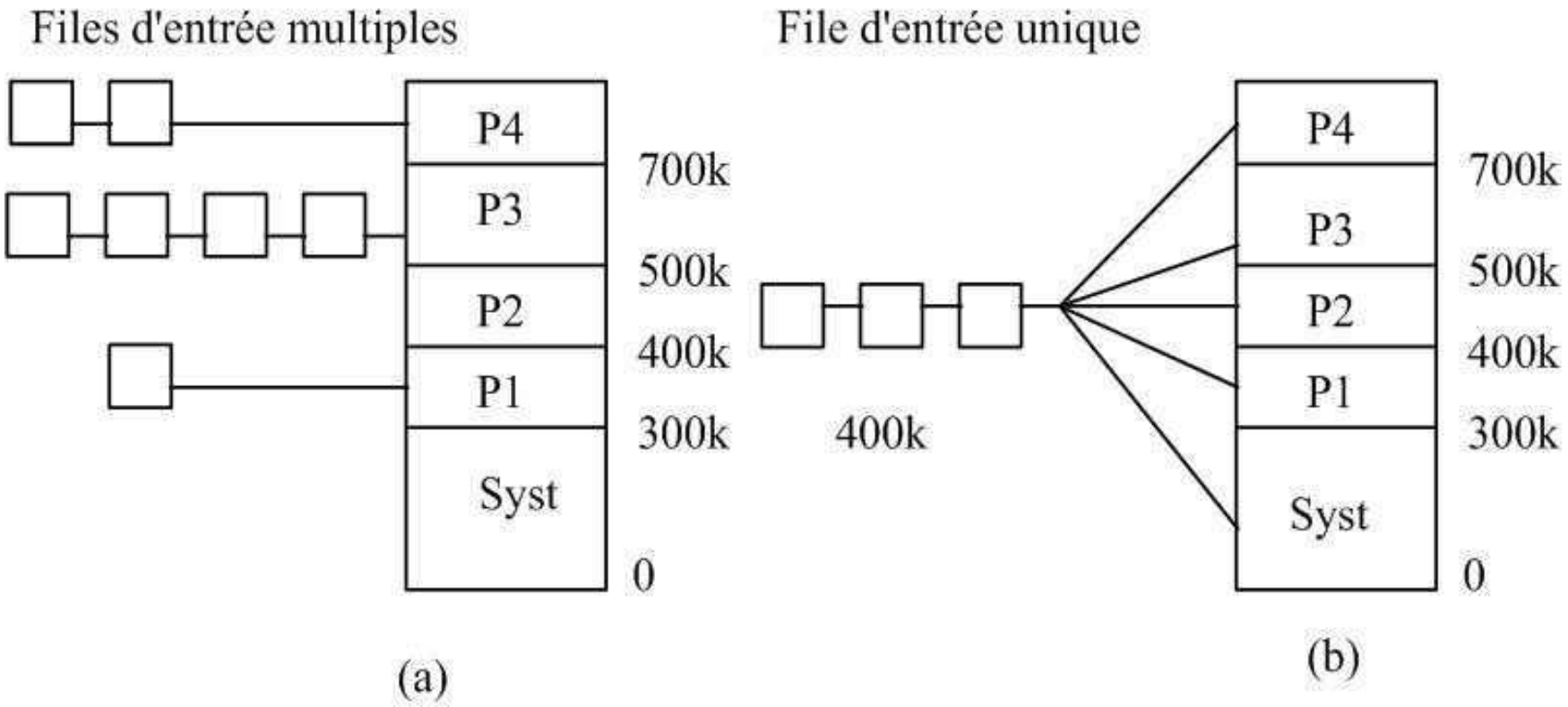
Adresse logique vs adresse physique:

- Une adresse mémoire physique : une “case” dans la mémoire centrale ;
- Une adresse mémoire logique : une adresse utilisée par un programme et calculée lors de la compilation.

ALLOCATION D'UNE ZONE MÉMOIRE CONTIGÜE

Allocation de partitions fixes:

Division de la mémoire en N partitions (pas forcément de tailles égales)



ALLOCATION D'UNE ZONE MÉMOIRE CONTIGÜE

Allocation de Partitions variables :

- Nombre et taille des processus varient au cours du temps ;
- Améliore l'usage de la mémoire MAIS complique son allocation et sa libération.

➔ Solution = compactage, mais ceci demande beaucoup de temps

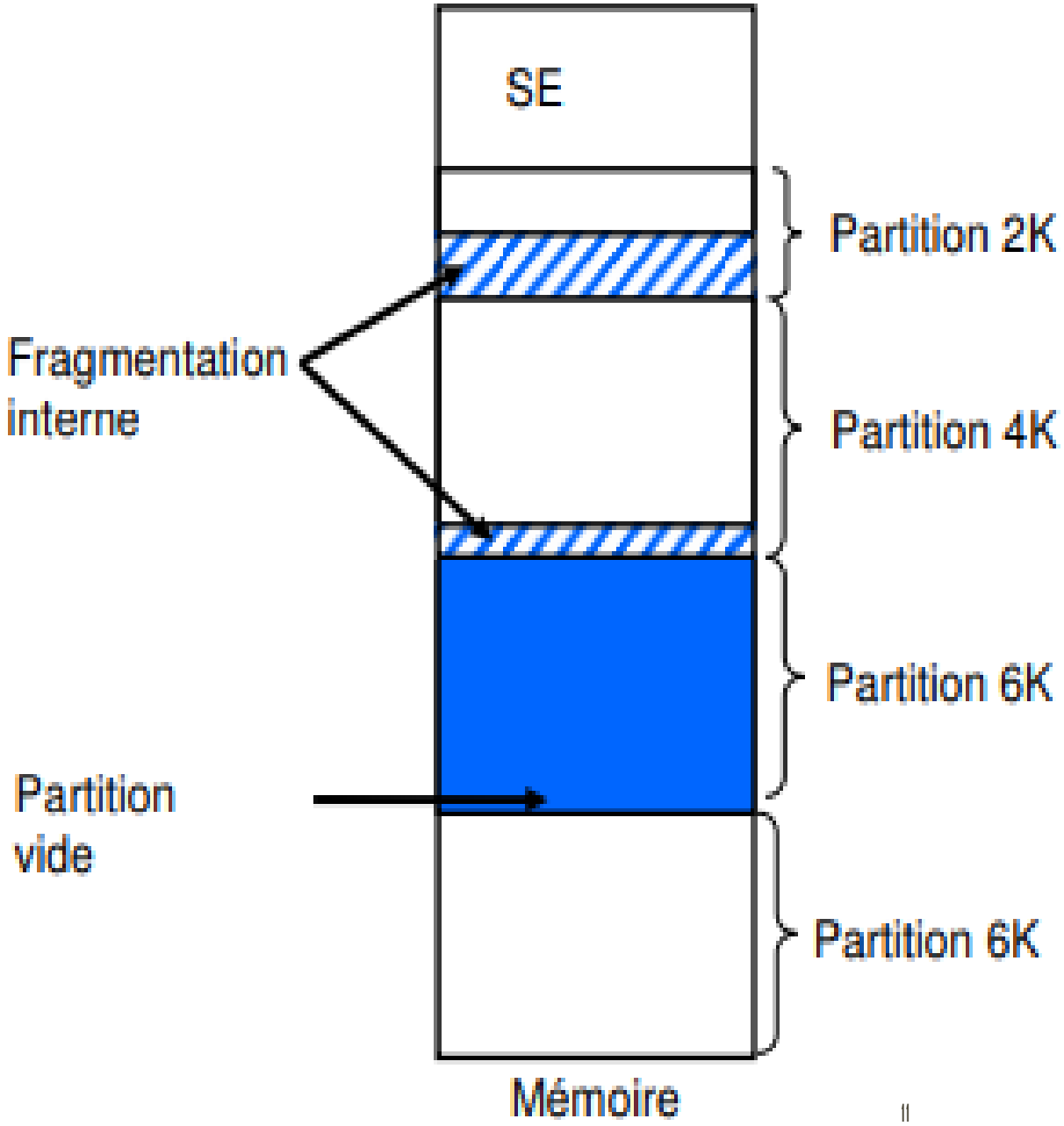
QUESTIONS:

Expliquez ce qu’est la **fragmentation** de la mémoire.

Dans un cadre d’allocation de la mémoire contiguë, de quelle façon peut on apporter remède à la fragmentation ?

Problème : Fragmentation interne

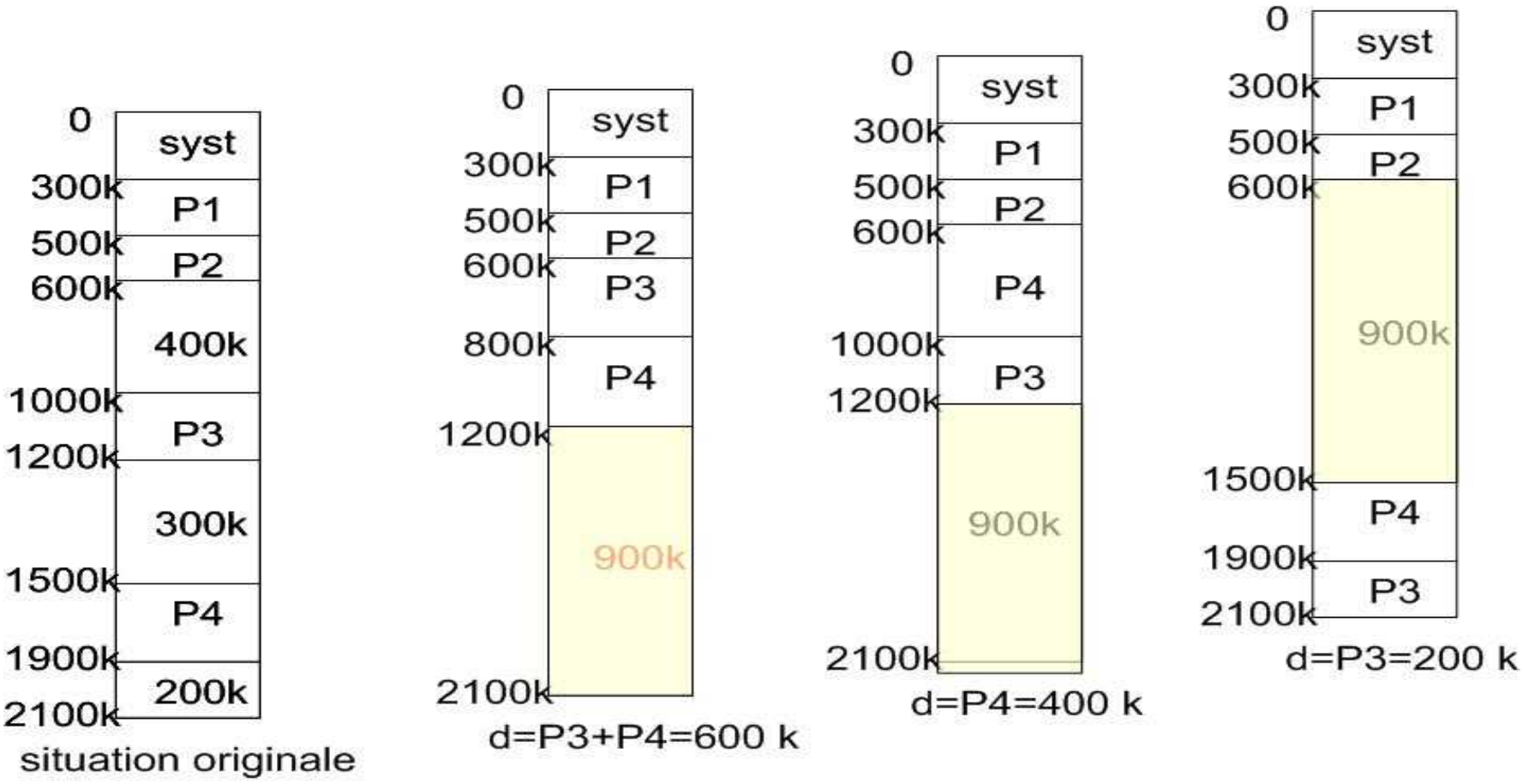
- *Fragmentation interne est* causée par la partie d'une partition non utilisée par un processus.



ALLOCATION D'UNE ZONE MÉMOIRE CONTIGÜE

Compactage:

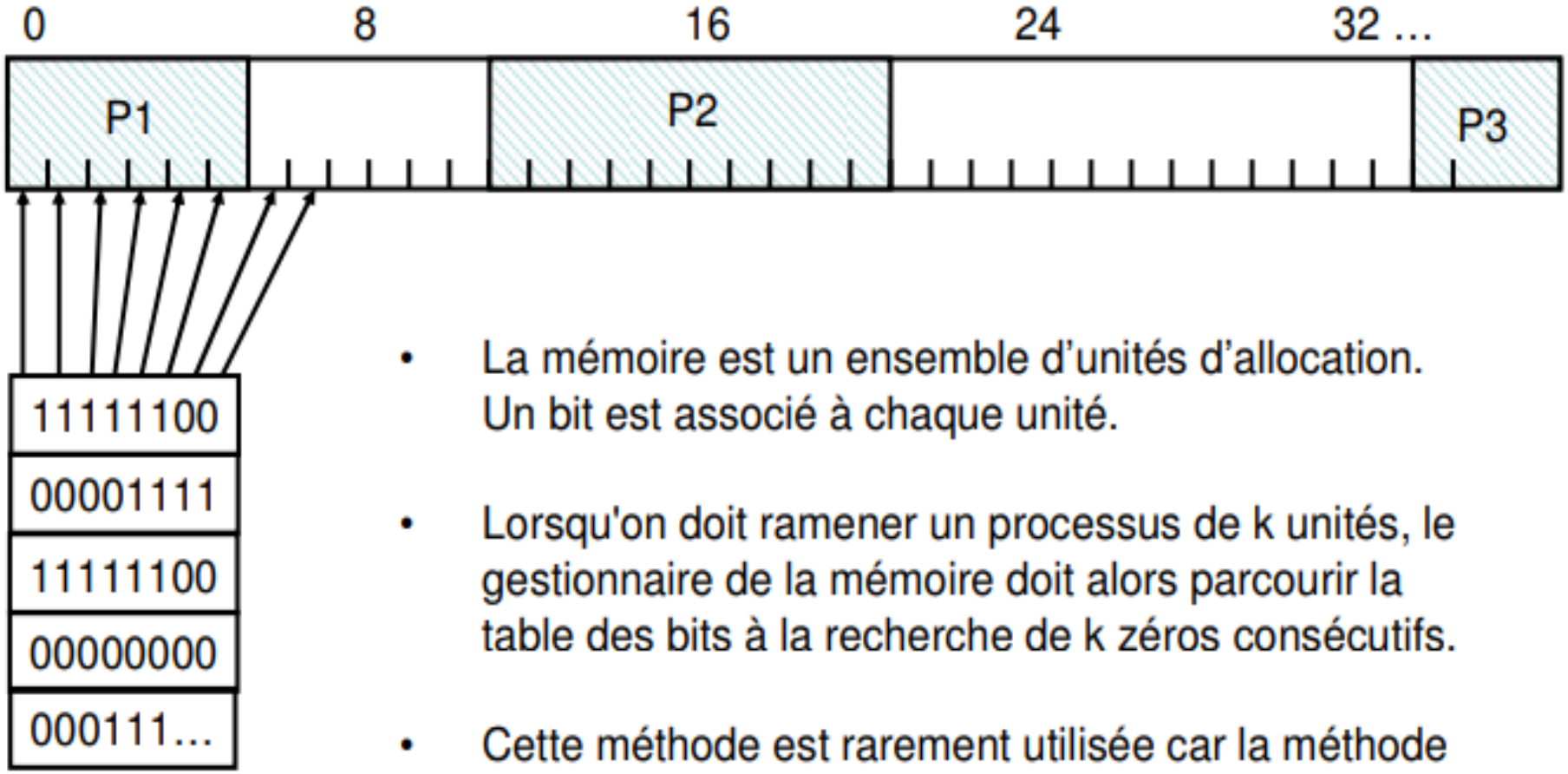
déplacer tous les processus vers une extrémité de la mémoire.



État de la mémoire

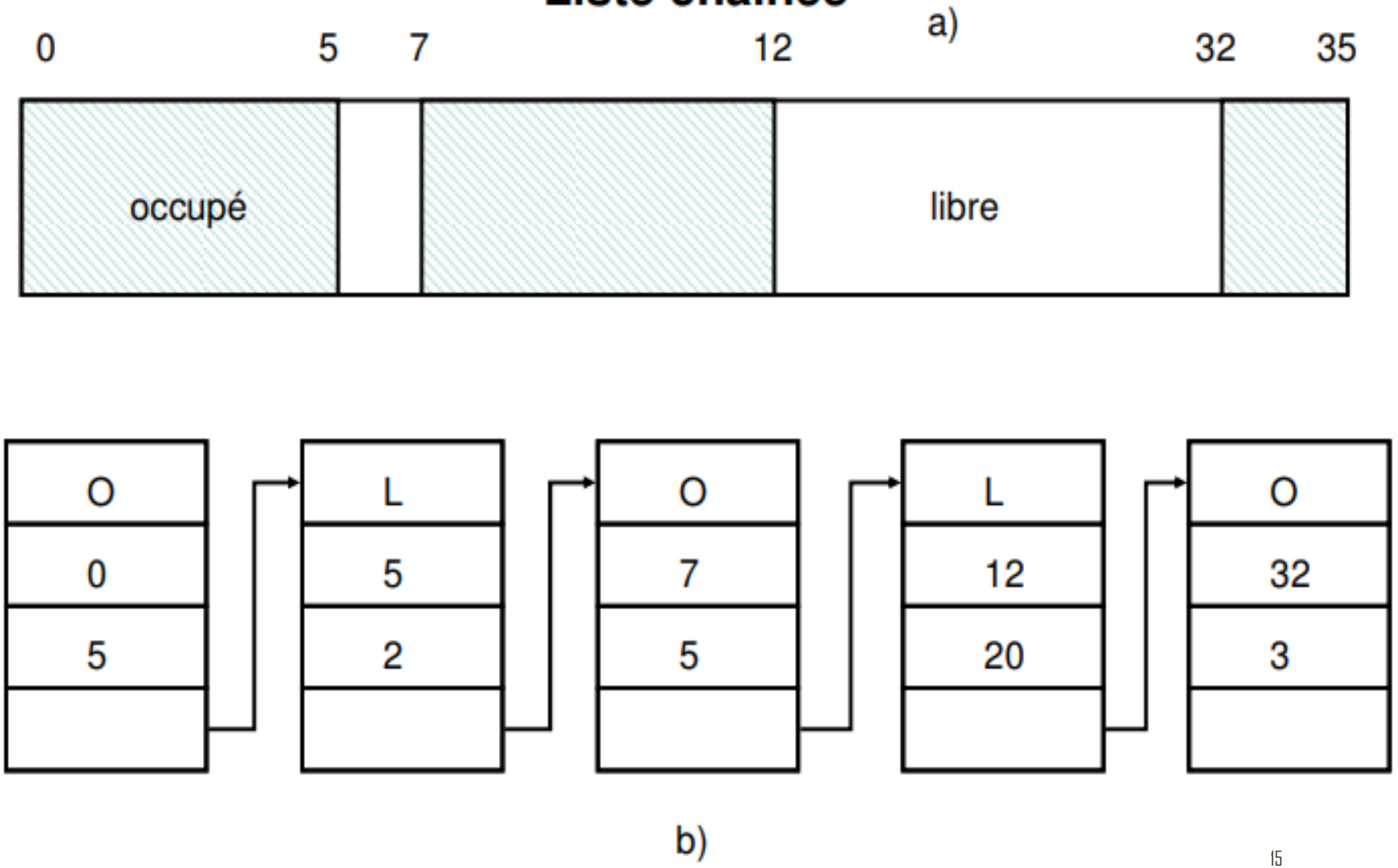
- Pour gérer l'allocation et la libération de l'espace mémoire, le gestionnaire doit :
  - Connaître l'état de la mémoire :
    - Tables de bits,
    - Listes chaînées.
  - Avoir une politique de placement et de récupération d'espace.
    - Premier ajustement
    - Meilleur ajustement
    - Pire ajustement
    - Par subdivision

État de la mémoire (2)
Table de bits (Bitmaps)

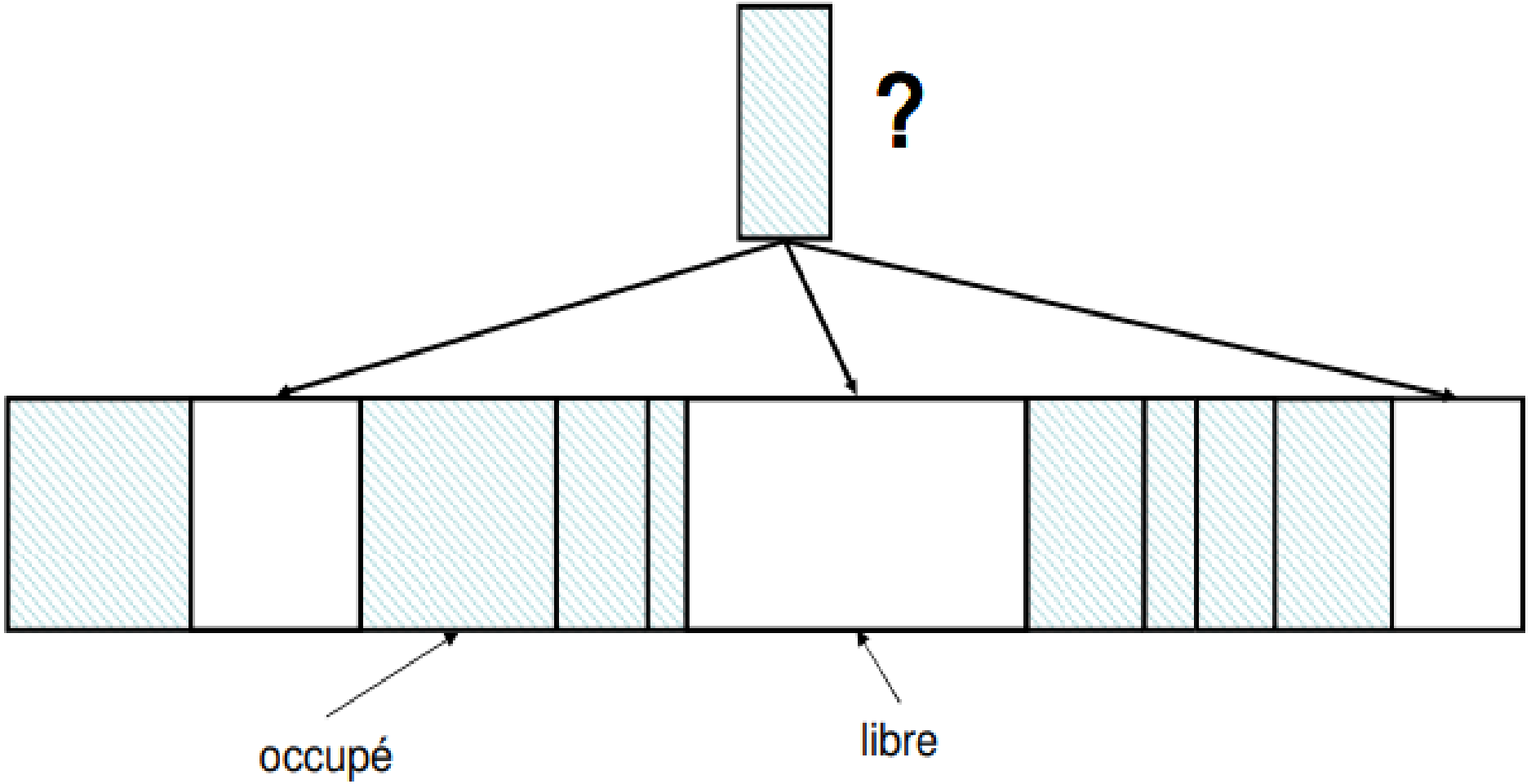


- La mémoire est un ensemble d'unités d'allocation. Un bit est associé à chaque unité.
- Lorsqu'on doit ramener un processus de k unités, le gestionnaire de la mémoire doit alors parcourir la table des bits à la recherche de k zéros consécutifs.
- Cette méthode est rarement utilisée car la méthode de recherche est lente (k zéros consécutifs).

État de la mémoire (3)
Liste chaînée



Politiques de placement



TECHNIQUES D'ALLOCATION DE LA MÉMOIRE

1. Première zone libre (**first fit**)
   - On trouve le premier trou suffisamment grand pour contenir le processus.
   - Le trou est ensuite divisé en deux parties: une pour le processus et l'autre pour la mémoire inutilisée.

2. Zone libre suivante (**next fit**)
   - Identique à l'algorithme précédent sauf que la recherche commence au dernier espace libre trouvé.
   - Les performance sont légèrement meilleures

TECHNIQUES D'ALLOCATION DE LA MÉMOIRE

3. Meilleur ajustement (**best fit**)
   - Parcourt toute la liste et recherche le plus petit trou pouvant contenir le processus.
   - Évite de partitionner inutilement les gros trous
   - Cet algorithme crée des trous minuscules inutilisables par la suite
   - Plus lent que les algorithmes précédents

4. Plus grand résidu (**worst fit**)
   - On prend le plus grand trou disponible
   - Le trou restant est assez grand pour être réutilisé
   - Des simulations démontrent que cette solution n'est pas meilleure que la précédente.

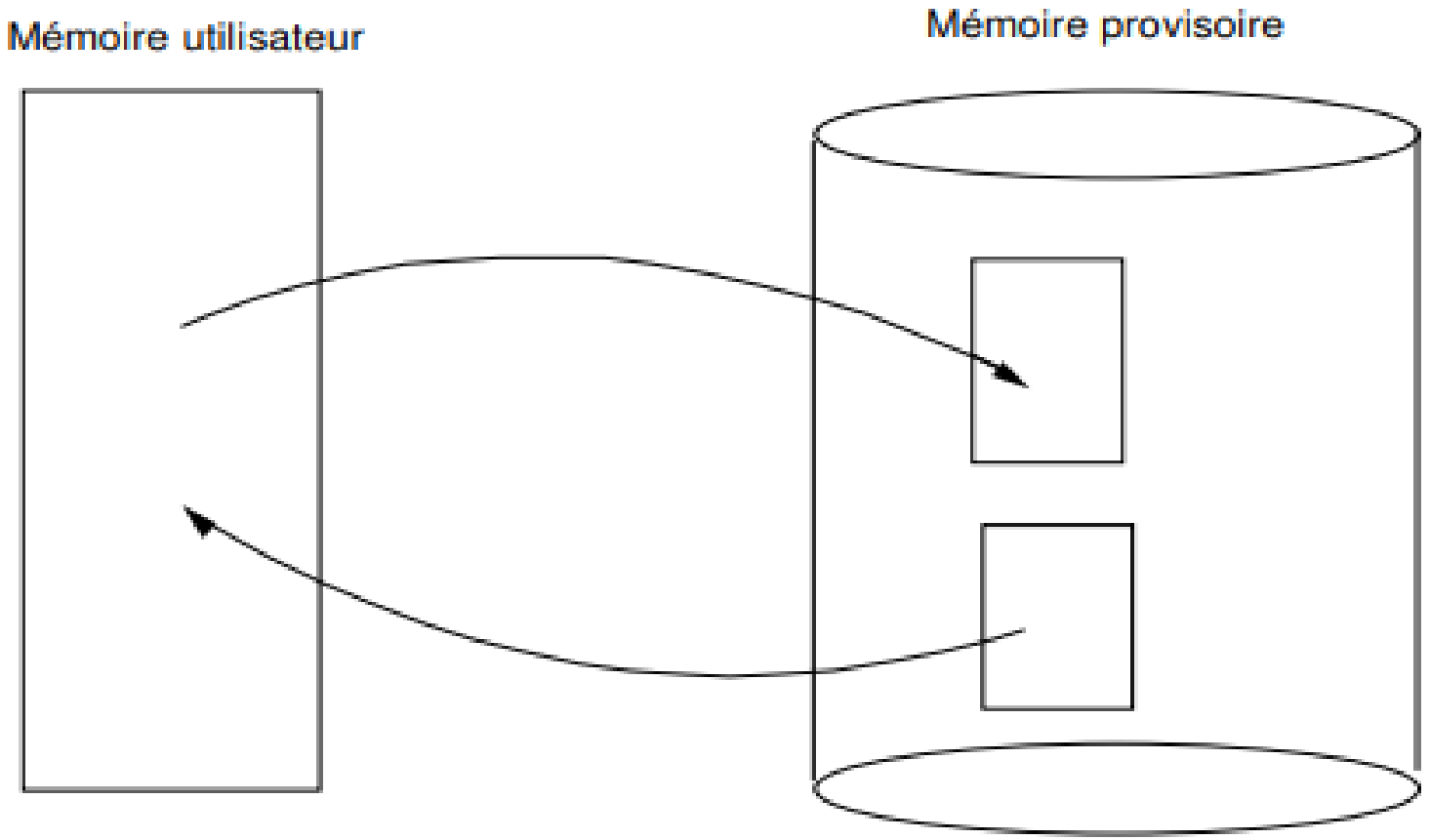
Politiques de placement (4)
Par subdivision

- Les tailles des blocs sont des puissances de 2.
- Initialement, on a un seul bloc libre.
- Supposons que la taille maximale est de 1 Mo et qu'un processus de 70 Ko demande à être chargé en mémoire.
- Le gestionnaire détermine d'abord la taille du bloc à allouer (égale à la plus petite puissance de 2 supérieure à 70Ko, soit 128 Ko).
- Comme il n'y a pas de blocs libres de taille 128 Ko, 256 Ko ou 512 Ko, la mémoire de 1 Mo est divisée en deux blocs de 512 Ko.
- Le premier bloc est divisé en deux blocs de 256 Ko. Enfin, le premier des deux blocs nouvellement créés est divisé en deux blocs de 128 Ko.
- L'espace alloué au processus est situé entre l'adresse 0 et 128 Ko.
- L'allocation par subdivision est rapide mais elle est assez inefficace en ce qui concerne l'utilisation de la mémoire (arrondir les tailles à une puissance de 2, fragmentation interne).

Le va-et-vient (Swapping)

- Le va-et-vient est mis en œuvre lorsque tous les processus ne peuvent pas tenir simultanément en mémoire.
- On doit alors en déplacer temporairement certains sur une mémoire provisoire, en général, une partie réservée du disque (mémoire de réserve, *swap area* ou *backing store*).
- Sur le disque, la zone de va-et-vient d'un processus peut être allouée à la demande. Quand un processus est déchargé de la mémoire centrale, on lui recherche une place.
- Les places de va-et-vient sont gérées de la même manière que la mémoire centrale.
- La zone de va-et-vient d'un processus peut aussi être allouée une fois pour toute au début de l'exécution. Lors du déchargement, le processus est sûr d'avoir une zone d'attente libre sur le disque (éviter des interblocages).

Le va-et-vient (Swapping) (2)



Exercice 1 :

On considère à l' instant T l'état suivant de la mémoire centrale d'un processus :

Espace libre	10k		20k			10k		30k		5k		20k	
Espace occupé		10k			30k		5k		10k		15k		10k

Des demandes d'allocation de mémoire arrivent dans l'ordre suivant :

D1 (20 Ko), D2 (10 Ko), D3 (5 Ko) et D4 (25 Ko).

A quelles emplacement mémoire sont alloues les blocs si on utilise les algorithmes d'allocation suivants :

1. Première zone libre (First Fit)
2. Meilleur ajustement (Best Fit)
3. Plus grand résidu (Worst Fit)
4. Zone libre suivante (Next Fit)

Exercice 2 :

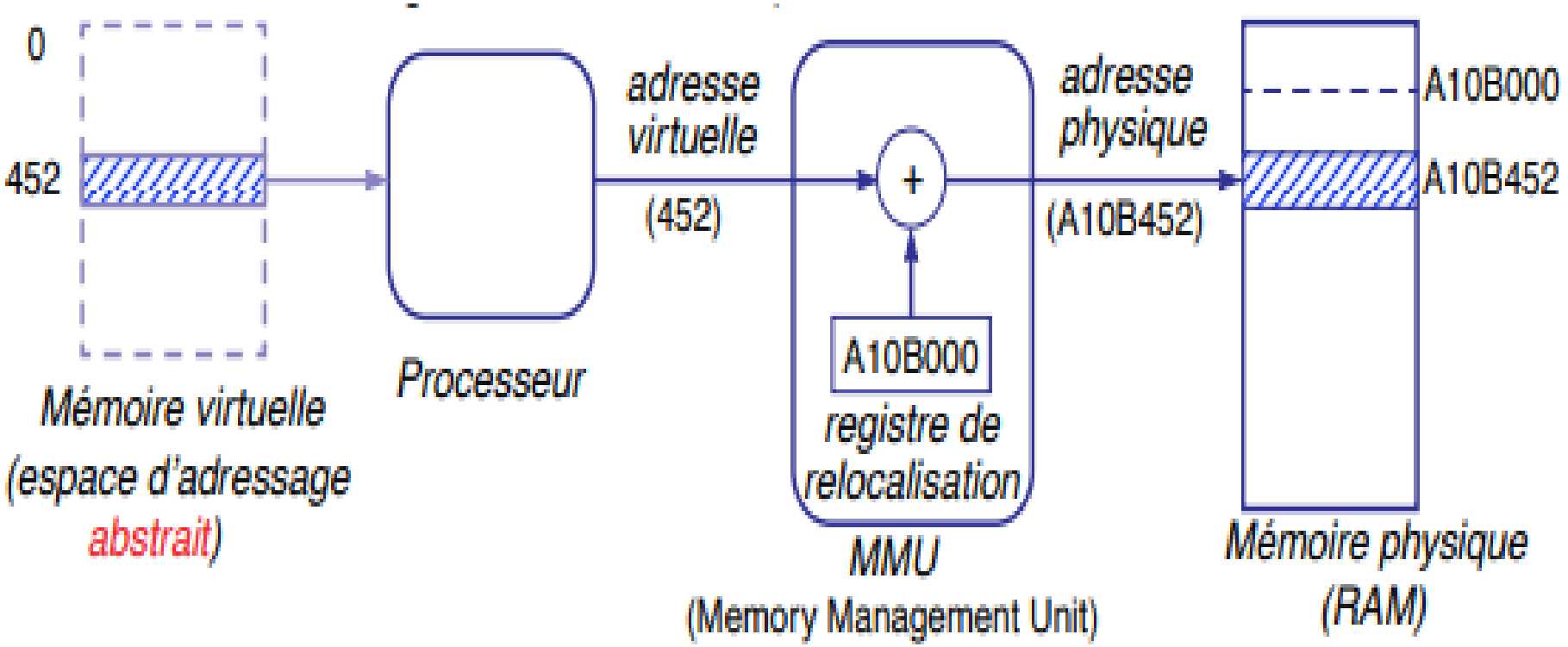
On suppose que la situation initiale de la mémoire est telle que représentée sur le schéma suivant :

Espace occupé	10k		20k			10k		30k		5k		20k	
Espace Libre		10k			30k		5k		10k		15k		10k

Soit le workload est le suivant :

1. Allocation d'un bloc G (20ko)
2. Libération du bloc B
3. Allocation d'un bloc H (15ko)
4. Libération du bloc E
5. Allocation d'un bloc I (40ko)
6. Allocation d'un bloc J (5ko)
7. Libération du bloc C
8. Allocation d'un bloc K (15ko)

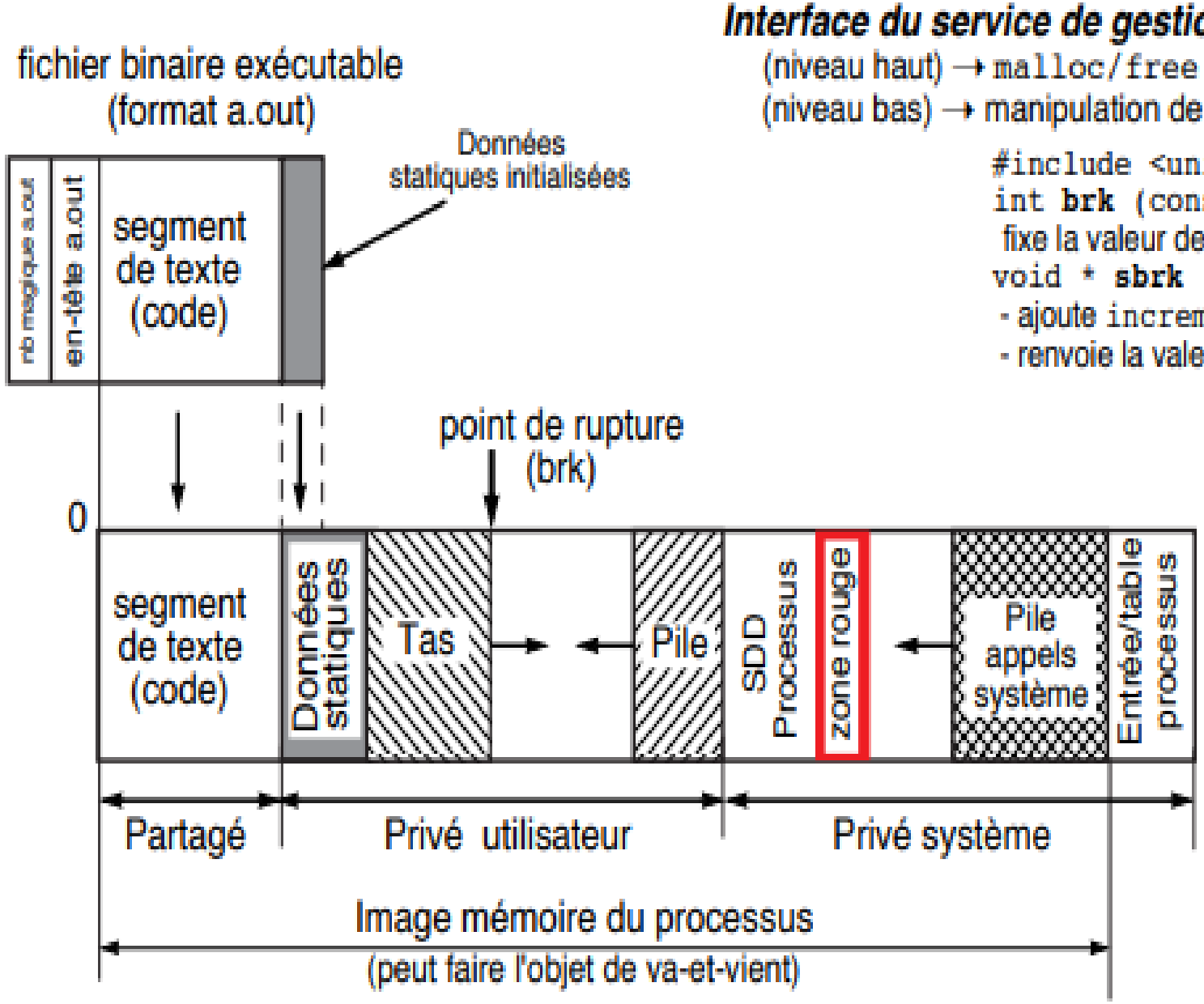
Étudier le comportement des algorithmes ''Best Fit" et ''Next Fit" en d'indiquant les éventuelles allocations qui échouent et de dessiner la situation de la mémoire à la fin du workload.



→ si le programme est déplacé lors de l'exécution, il suffit d'actualiser le registre de relocalisation

→ l'interprétation des adresses virtuelles facilite aussi la protection de la mémoire

Exemple : espace d'adressage virtuel d'un processus UNIX



Interface du service de gestion mémoire :

(niveau haut) → `malloc/free (stdlib.h)`

(niveau bas) → manipulation de `brk` :

```
#include <unistd.h>
int brk (const void *ptr)
 fixe la valeur de brk
void * sbrk (int increment)
 - ajoute increment
 - renvoie la valeur précédente de brk
```

3) Allocation fragmentée

a) Pagination (Allocation dynamique et fragmentée de zones fixes)

Idée : fractionner l'image mémoire d'un processus

(et gérer la correspondance entre ces fragments et leur implantation en mémoire physique)

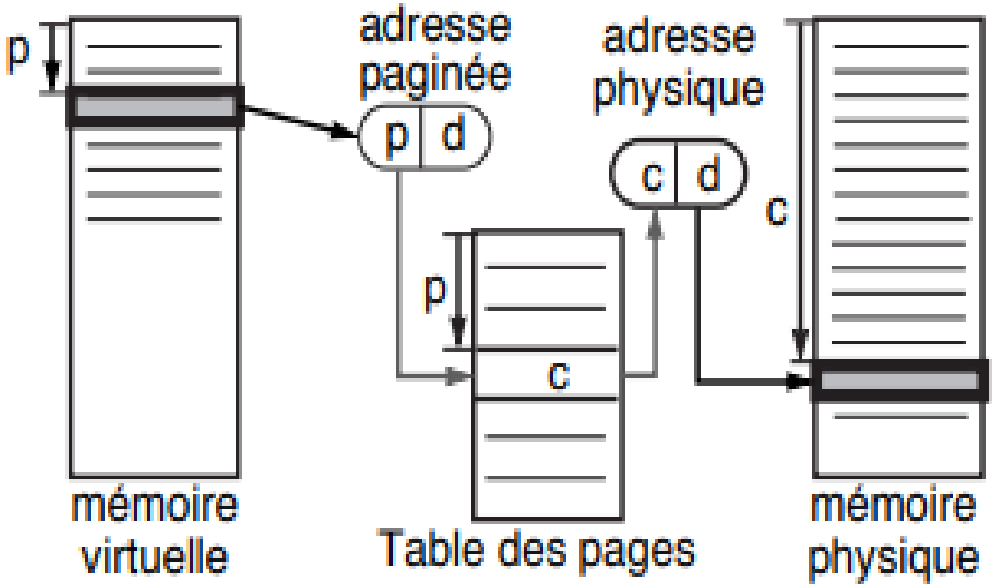
Pages et cases

- La mémoire *virtuelle* est découpée en *pages*
- La mémoire *physique* est découpée en *cases*
- Cases et pages ont une *taille* P *fixée* et *identique*

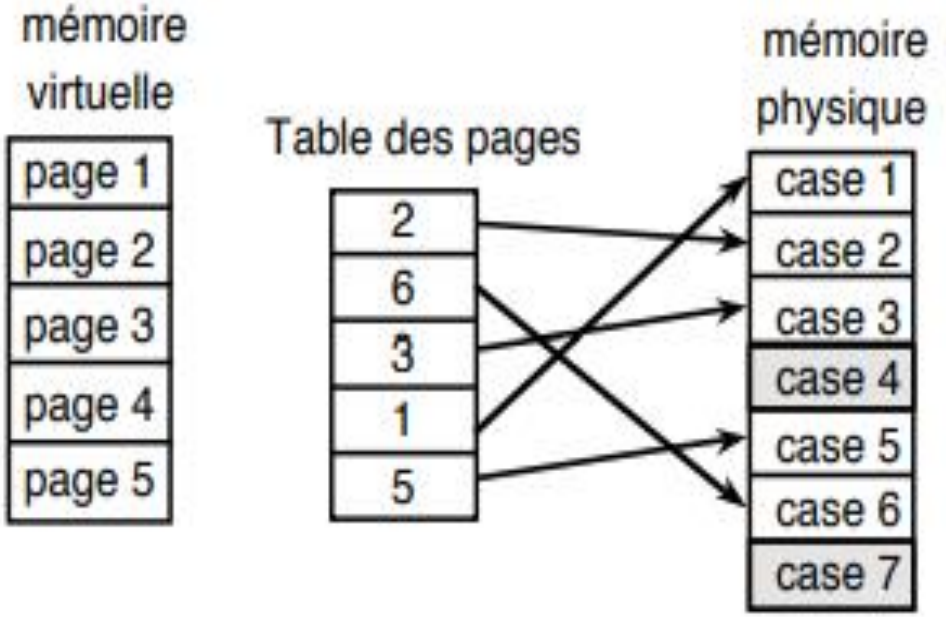
Evaluation de l'adresse virtuelle (A_v)

A_v = (n° de page,déplacement dans la page) = (p,d),

avec p = A_v div P et d = A_v mod P (efficace avec P= 2^k)



Exemple



Avantages

- pas de fragmentation externe
- fragmentation interne : 1/2 page par processus, en moyenne
  → ne pas choisir des pages trop volumineuses (généralement 2K-8K)
- adressage indirect – possibilité de
  - partager une page
  - contrôler l'accès à une page

b) Segmentation (Allocation dynamique et fragmentée de zones variables)

Mémoire vue comme un *ensemble* de zones (*segments*)

- de longueurs variables
- sans ordre relatif

Principe

Un segment regroupe des éléments homogènes (du point de vue de leur utilisation)

→ contrôles et protection fins :

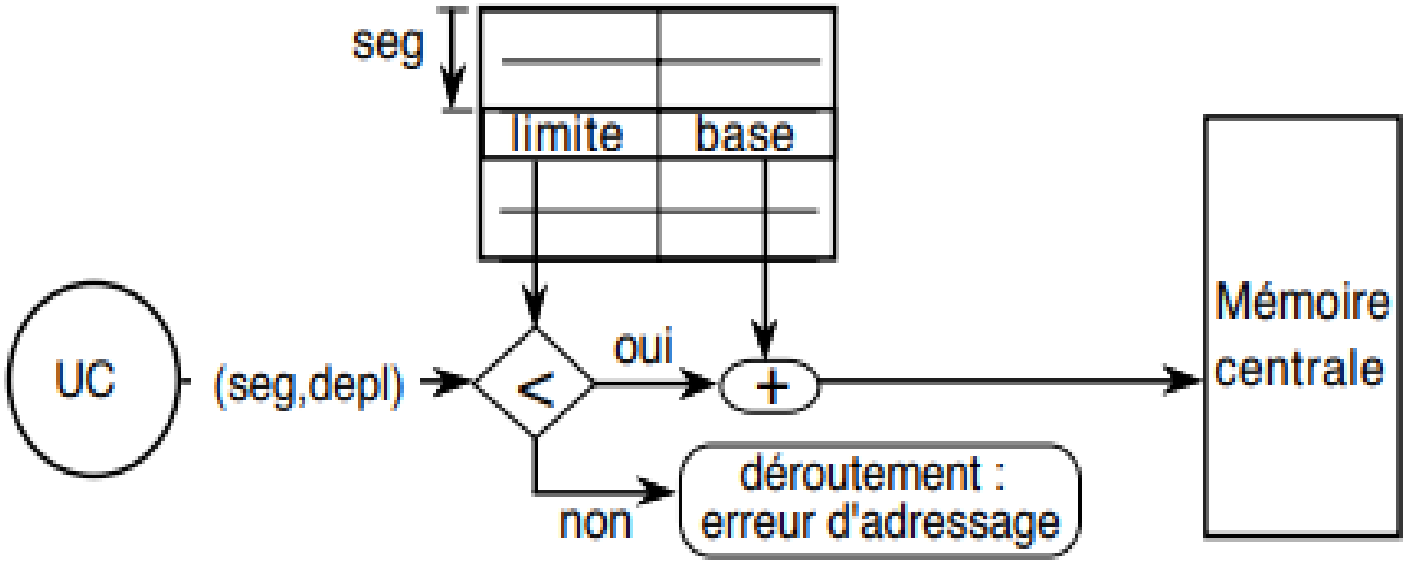
Pour chaque segment

- contrôle des *accès* (opérations spécifiques aux objets du segment)
- contrôle de l'*adressage*
  Exemple : si le segment est un tableau d'objets, on conserve sa dimension

Mise en œuvre

adresse segmentée : (id. de segment, déplacement dans le segment)

Mécanisme : table des segments



Mise en œuvre de la table des segments

Mêmes mécanismes que table des pages : table en mémoire, registres associatifs

Fragmentation externe

Le mécanisme de réimplantation dynamique facilite le recompactage, qui reste coûteux.

Partage de segments

Référence en mémoire physique commune à deux tables de segments

→ mise en commun de données, utilitaires, programmes

c) Segmentation paginée

But : conserver l'abstraction donnée par la segmentation, et éviter la fragmentation externe

Idée : paginer les segments

Principe

- une table des segments pour chaque processus
- une table des pages pour chaque segment

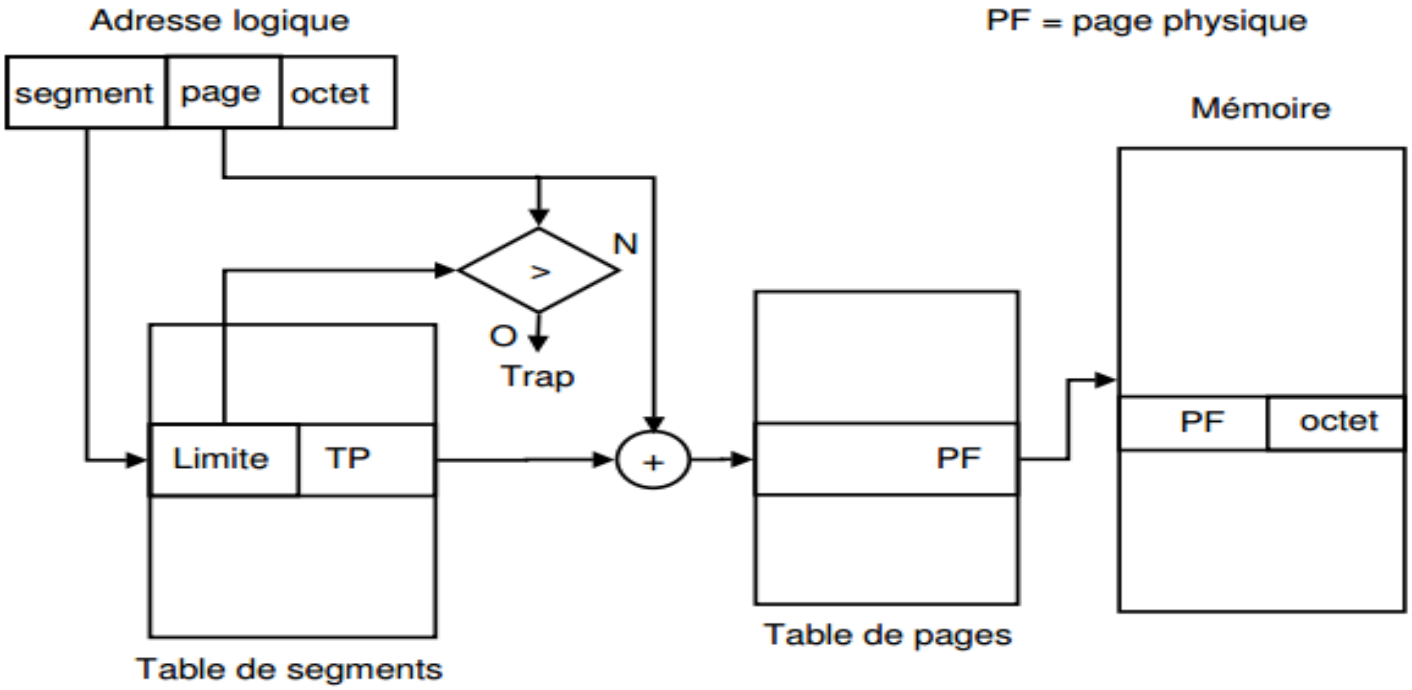


FIG. 10.15 – Segmentation paginée.

a) Intégration du va-et-vient et de la pagination

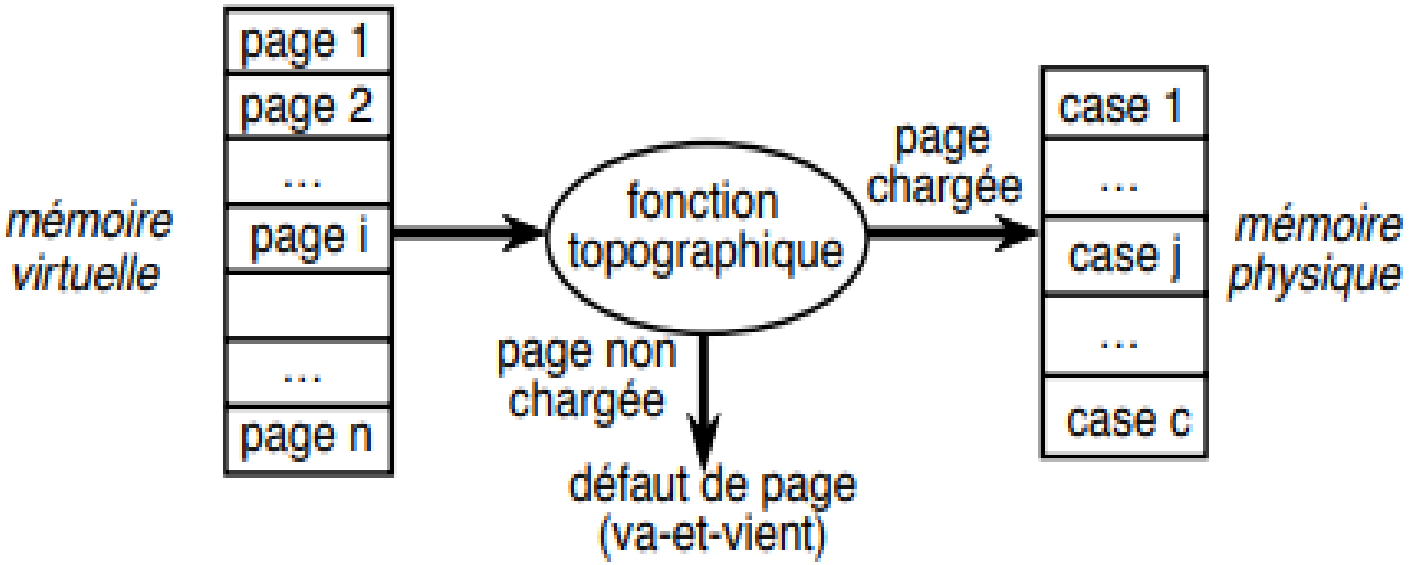
Idée : appliquer aux pages le principe du va-et-vient

→ il suffit d'ajouter à chaque entrée de la table des pages un *indicateur de présence* en mémoire

Motivation : principe de localité (le passé récent est une bonne image du futur proche)

→ l'ensemble des pages utilisées par un processsus est petit et plutôt stable → va-et-vient limité

Principe de fonctionnement



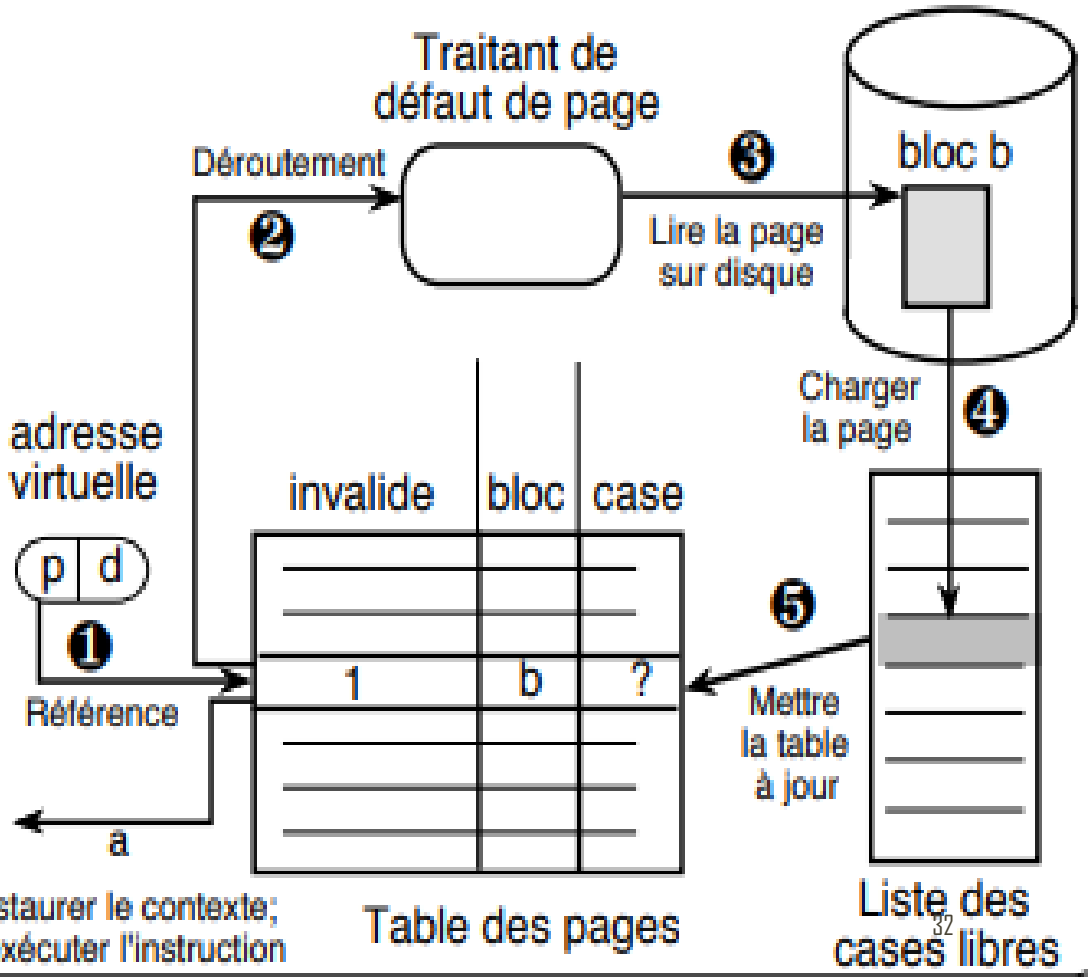
Remarques

- L'emploi de zones fixes simplifie le placement : toutes les cases se valent
- La fonction topographique est généralement implantée sous la forme d'une table des pages

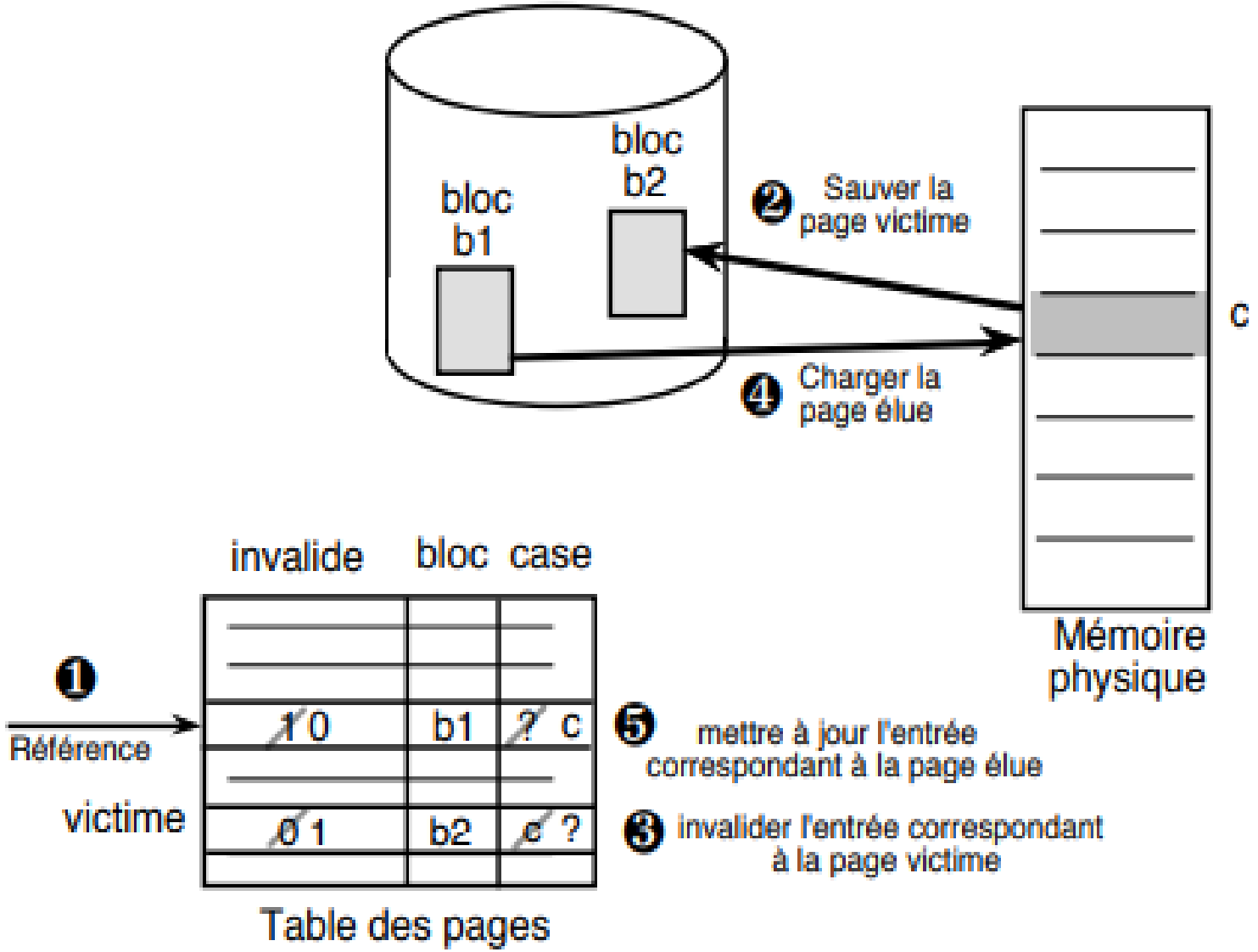
Mécanisme de défaut de page

- repérer les cases libres ou libérables → liste des cases libres
- ajout d'une colonne (*invalide*) à la table des pages,
  indiquant pour chaque page si elle est chargée en mémoire
- connaître l'adresse de chaque page en mémoire secondaire
  Cas usuel : les blocs (disque) sont de la même taille que les pages
  → ajout d'une colonne ("*bloc*") à la table des pages [≈ *table de couplage* (n° page, n° bloc)]

Mécanisme de chargement



Va-et-vient « page à page » si la liste des cases libres est vide



Remarque : le nombre d'E/S causées par un défaut de page est doublé

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	2	2	2	2	2	2	2	2	2	2	**7**	7	7
1		**0**	0	0	0	0	0	**4**	4	4	**0**	0	0	0	0	0	0	0	0	0
2			**1**	1	1	**3**	3	3	3	3	3	3	3	**1**	1	1	1	1	1	1

FIG. 10.9 – Algorithme optimale de Belady.

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	0	0	0	0	**7**	7	7
1		**0**	0	0	0	**3**	3	3	**2**	2	2	2	2	**1**	1	1	1	1	**0**	0
2			**1**	1	1	1	**0**	0	0	**3**	3	3	3	3	**2**	2	2	2	2	**1**

FIG. 10.10 – Algorithme de remplacement FIFO.

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	1	1	1	1	**1**	1	1
1		**0**	0	0	0	0	0	**0**	0	**3**	3	3	3	3	3	**0**	0	0	0	0
2			**1**	1	1	**3**	3	3	**2**	2	2	2	2	2	2	2	2	**7**	7	7

FIG. 10.11 – Algorithme de remplacement de la page la moins récemment utilisée.

Gestion d'un vaste espace virtuel

Problème : taille potentielle de la table des pages

Solution1 : liste inverse de pages (liste de cases)

Liste par case : < id de processus, n° de page >

- liste des cases réduite → accès séquentiel
- sinon, accès calculé (fonction de dispersion)
- usage courant d'une mémoire associative

Solution 2 : pagination hiérarchique (hyperpages)

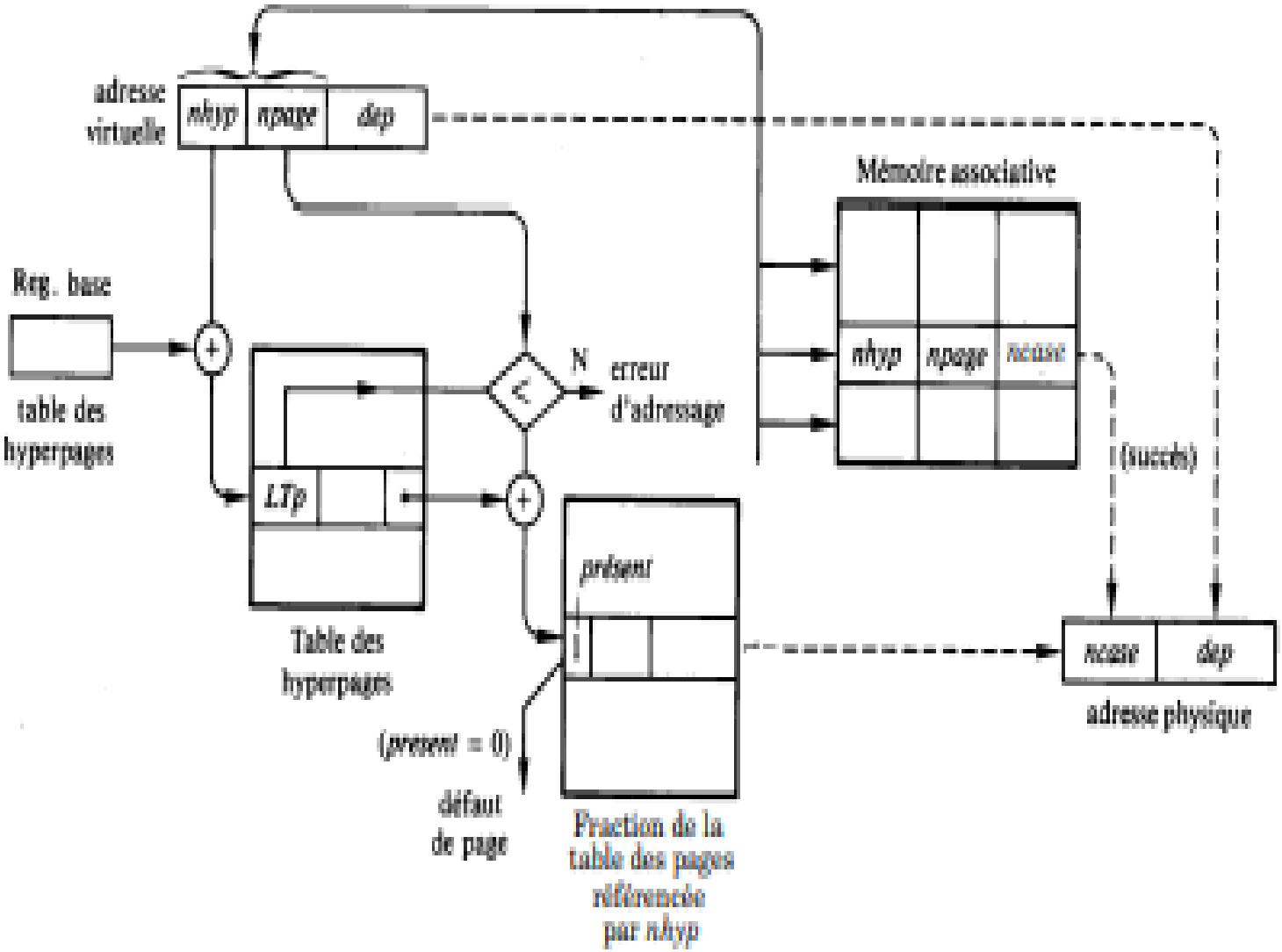
Idée : paginer la table des pages

- la table des pages est découpée en hyperpages
- la table des hyperpages (THP) permet de ne désigner que les parties de la table des pages effectivement utilisées
  → un compteur "nombre de pages désignées" est associé à chaque entrée de la THP
- une adresse virtuelle devient : (n°hyperpage, n°page, déplacement)

Remarques

- Mémoire associative nécessaire
- Pas de défaut de page pour la table des hyperpages !
  Exemple : VM 370 → pages : 2 ou 4K ; hyperpages 64 octets ou 1K
- Ce principe peut être réitéré → hiérarchie d'hyperpages (Linux : 3 niveaux prévus)

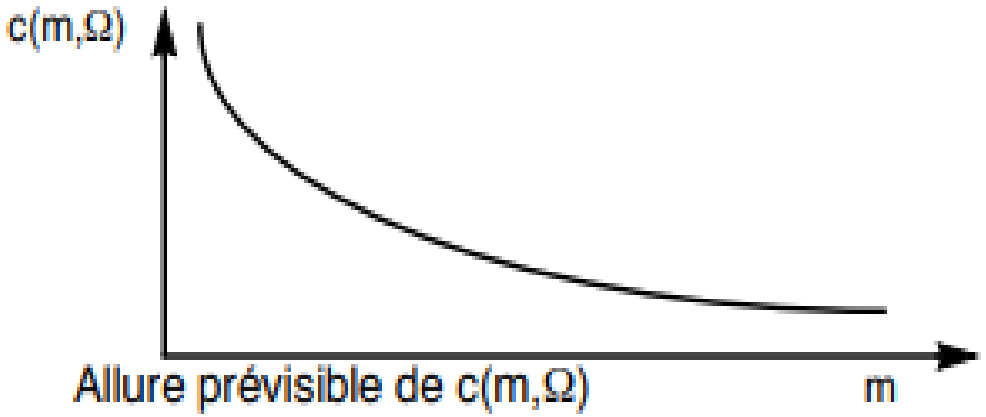
Pagination à 2 niveaux (S. Krakowiak, principes des systèmes d'exploitation (Dunod), p362)



d) Choix des pages victimes : algorithmes de remplacement

Critères d'évaluation du comportement d'un processus/algorithme

- c(m,Ω) : *nombre de chargements de pages* causés par Ω
  - ◊ Ω : suite de références (numéros de pages virtuelles demandées par le processus),
  - ◊ m : taille de la mémoire allouée au processus, en pages



Remarque : certaines stratégies de remplacement présentent des « anomalies »

Exemple (Belady) :

- gestion FIFO des requêtes de remplacement
- Pour Ω = 1,2,3,4,1,2,5,1,2,3,4,5 on a : c(3,Ω) < c(4,Ω)

• *taux de défaut de pages* $F = \frac{c(m,\Omega)}{|\Omega|}$ (|Ω| : longueur de la suite de références)

Approximation de la solution optimale : LRU

Algorithme optimal

Remplacer la page dont la prochaine référence est la plus éloignée dans le temps

→ évaluer a priori la chaîne de référence

→ irréalisable en général

Ordre chronologique de chargement (LRU , Least Recently Used)

Idée : le principe de localité est une approximation de la stratégie "optimale"
(le passé récent est une bonne image du futur proche)

→ la victime sera la page qui n'a pas été utilisée depuis le plus longtemps

Remarques

- LRU ne présente pas l'anomalie de Belady
- La mise en œuvre directe de LRU est coûteuse (activée à chaque référence)
  - ◊ gérer une « pile » : liste doublement chaînée, avec pointeurs de tête et de queue.
    Fonctionnement : à chaque nouvelle référence, on place la page référencée en tête de pile.
  - ◊ numéroter les références successives → associer un compteur (*date logique*) à chaque page
    Difficultés : débordement du compteur, arithmétique

Mise en œuvre approchée de LRU

utilisent un bit de référence (0 initialement, mis à 1 lors de l'accès) associé à chaque page

Représentation « économique » du compteur de date logique

Mot contenant l'historique des valeurs des bits de référence sur une période de temps :

- le bit de gauche est le bit de référence courant
- le mot est décalé vers la droite, à intervalles réguliers
- la plus petite valeur correspond à l'accès le moins récent

0	0	1	1	
0	1	0	0	
0	1	1	0	
0	0	0	1	
0	0	0	0	

Algorithme de la deuxième chance

- La victime est recherchée en parcourant la table des pages de manière circulaire
- L'entrée courante de la table des page a un bit de référence valant
  - ◊ 1 → bit remis à 0 ; la page est placée en queue de file
  - ◊ 0 → c'est la victime